

4 Advanced JSF

Nachdem in den vorangegangenen Kapiteln die Grundlagen von JSF im Mittelpunkt standen, wollen wir uns in diesem Kapitel den etwas weiterführenden Themen zuwenden.

Nach einer kurzen Vorstellung der in JSF 2.0 neu hinzugekommenen Project-Stage in Abschnitt 4.1 zeigen wir Ihnen in Abschnitt 4.2 erweiterte Aspekte von Facelets, die hauptsächlich die Wiederverwendung von Inhalten betreffen. Ein zentrales Thema für beinahe jedes Webprojekt ist Templating, das mit der Integration von Facelets nun endlich auch in den JSF-Standard Einzug gehalten hat. Abschnitt 4.3 zeigt, wie Templating mit Facelets funktioniert. Abschnitt 4.4 präsentiert ein weiteres, lang ersehntes Feature von JSF 2.0: die Unterstützung von GET-Anfragen und View-Parametern. Anschließend folgt in Abschnitt 4.5 eine kurze Einführung in die ebenfalls mit JSF 2.0 eingeführten Ressourcen. Abschließend werfen wir in Abschnitt 4.6 noch einen etwas ausführlicheren Blick auf den Faces-Context und den External-Context, bevor wir das Kapitel mit einigen Details der Konfiguration von JSF in Abschnitt 4.7 abschließen.

Damit die Praxis nicht zu kurz kommt, werden die vorgestellten Konzepte in den Beispielen *MyGourmet 10*, *MyGourmet 11* und *MyGourmet 12* umgesetzt.

4.1 Project-Stage

Die in JSF 2.0 eingeführte Project-Stage ist an `RAILS_ENV` von *Ruby on Rails* angelehnt und bietet eine Möglichkeit, die aktuelle Phase des Projekts für die Entwicklung bereitzustellen. Die möglichen Werte sind in der Enum `javax.faces.application.ProjectStage` festgelegt und lauten folgendermaßen:

- Production (Standardwert)
- Development
- SystemTest
- UnitTest

Die Projektphase kann auf folgende Arten auf einen der oben angeführten Werte gesetzt werden:

- ❑ Über den Kontextparameter `javax.faces.PROJECT_STAGE` in der `web.xml`
- ❑ Über den Namen `java:comp/env/jsf/ProjectStage` mit JNDI

Mit dem Ausschnitt aus der `web.xml` in Listing 4.1 wird die Project-Stage zum Beispiel auf den Wert `Development` gesetzt.

Listing 4.1
Project-Stage in
`web.xml` setzen

```
<context-param>
  <param-name>javax.faces.PROJECT_STAGE</param-name>
  <param-value>Development</param-value>
</context-param>
```

Zur Laufzeit wird die aktuelle Project-Stage im Application-Objekt abgeleitet und kann mit der Methode `getProjectStage()` von dort ausgelesen werden. Listing 4.2 zeigt ein kleines Codebeispiel.

Listing 4.2
Überprüfen der
Project-Stage
(Variante 1)

```
FacesContext fc = FacesContext.getCurrentInstance();
Application a = fc.getApplication();
if (a.getProjectStage() == ProjectStage.Development) {
    // Beliebiger Code
}
```

Mit der Hilfsmethode `isProjectStage(ProjectStage)` im `FacesContext` lässt sich das vorherige Codefragment noch weiter vereinfachen. In Listing 4.3 finden Sie ein Beispiel.

Listing 4.3
Überprüfen der
Project-Stage
(Variante 2)

```
FacesContext fc = FacesContext.getCurrentInstance();
if (fc.isProjectStage(ProjectStage.Development)) {
    // Beliebiger Code
}
```

Wo benötigen Sie die neue Project-Stage-Eigenschaft? Überall dort, wo Sie Code abhängig von der aktuellen Projektphase ausführen wollen. In JSF 2.0 wird das bereits in der Spezifikation an einigen Stellen berücksichtigt.

Wenn die Project-Stage auf `Development` gesetzt ist, wird in jede Seite eine `h:messages`-Komponente eingefügt, falls diese nicht vorhanden ist. Damit wird verhindert, dass Validierungsfehler in Formularen unbemerkt bleiben.

Ein weiteres Beispiel findet sich beim Ressourcenmanagement. JSF cacht Ressourcen nur dann, wenn die Project-Stage auf `Production` gesetzt ist. Andernfalls werden sie bei jedem Zugriff neu geladen.

Die *Project-Stage* ist eine der kleineren neuen Features in JSF 2.0, wird aber bereits an einigen Stellen verwendet und bietet viel Potenzial für weitere Einsatzgebiete.

4.2 Advanced Facelets

In allen bisherigen Beispielen haben wir Facelets nur als Seitendeklarationsprache und bessere Alternative zu JSP eingesetzt. Facelets kann aber viel mehr und bietet eine breite Palette an Features, die das Leben eines JSF-Entwicklers einfacher machen. Im Laufe dieses Abschnitts werden wir einige davon vorstellen.

Facelets stellt dazu eine eigene Tag-Bibliothek mit dem Namensraum `http://java.sun.com/jsf/facelets` bereit. Üblicherweise wird diese Bibliothek mit dem Präfix `ui` in Seitendeklarationen eingebunden. Die wichtigsten Tags daraus werden wir im Rest dieses Abschnitts präsentieren.

4.2.1 Wiederverwendung von Inhalten mit Facelets

Facelets bietet Entwicklern die Möglichkeit, Ansichten modular aufzubauen und wiederkehrende Inhalte an zentraler Stelle zu definieren. Das dafür grundlegende Konzept sind die sogenannten Kompositionen, die einen Teil eines Komponentenbaums gruppieren.

Facelets kann eine Ansicht aus einer beliebigen Anzahl von Kompositionen aufbauen, die jeweils in einem eigenen XHTML-Dokument deklariert sind. Trifft Facelets beim Aufbau des Komponentenbaums auf ein `ui:include`-Tag, wird das Dokument mit dem im Attribut `src` angegebenen Dateinamen in die Ansicht mit aufgenommen. Der Pfad des Dokuments kann absolut oder relativ zur aktuellen Ansicht angegeben sein.

Sehen wir uns das im Kontext von *MyGourmet* an. Listing 4.4 definiert eine Komposition für einen Seitenkopf, der unter `/WEB-INF/includes/header.xhtml` abgelegt ist.

In Listing 4.5 sehen Sie den Ausschnitt der Seitendeklaration `showCustomer.xhtml` mit dem über `ui:include` eingefügten Seitenkopf. Das Tag `ui:param` übergibt den Text für die Überschrift zweiter Ordnung als Parameter an das eingefügte Seitenfragment – doch dazu später mehr.

Wie funktioniert in Facelets die Zusammensetzung der Deklaration `showCustomer.xhtml` mit dem eingefügten Fragment `header.xhtml`? Wie Sie vielleicht bemerkt haben, handelt es sich bei beiden Dateien um komplette XHTML-Dokumente. Es soll allerdings nur ein einziges Dokument an den Browser geschickt werden. Des Rätsels Lösung

liegt darin, wie Facelets das Tag `ui:composition` behandelt – es ignoriert beim Einfügen des Seitenfragments sämtliche Inhalte außerhalb des Tags `ui:composition`.

Der Parameter `pageTitle` ermöglicht eine individuelle Definition der Überschrift bei jedem Einfügen des Fragments. Werfen Sie nochmals einen Blick auf Listing 4.4, dann sehen Sie die Verwendung dieses Parameters. Mit dem EL-Ausdruck `#{pageTitle}` wird der Wert direkt im `h2`-Element ausgewertet.

Listing 4.4

*Fragment für einen
Seitenkopf*

```

1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:h="http://java.sun.com/jsf/html"
3     xmlns:ui="http://java.sun.com/jsf/facelets">
4 <head><title>MyGourmet header</title></head>
5 <body>
6   <ui:composition>
7     <h:panelGroup style="width: 100%; height: 40px;"
8       layout="block">
9       <h:graphicImage value="/images/logo.png"
10        style="float: left;"/>
11       <h1 style="display: inline; margin-left: 5px;">
12         #{msgs.title_main}
13       </h1>
14     </h:panelGroup>
15     <h2>#{pageTitle}</h2>
16   </ui:composition>
17 </body>
18 </html>

```

Listing 4.5

*Einfügen des
Seitenkopfs mit
ui:include*

```

1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:f="http://java.sun.com/jsf/core"
3     xmlns:h="http://java.sun.com/jsf/html"
4     xmlns:ui="http://java.sun.com/jsf/facelets">
5 <head>
6   <title>#{msgs.title_main}</title>
7 </head>
8 <body>
9   <ui:include src="/WEB-INF/includes/header.xhtml">
10     <ui:param name="pageTitle"
11       value="#{msgs.title_show_customer}"/>
12   </ui:include>
13   ...
14 </body>
15 </html>

```

Das `ui:component`-Tag bietet die gleiche Funktionalität wie das Tag `ui:composition` – im Komponentenbaum wird aber zusätzlich eine Wurzelkomponente für die Komponentengruppe eingefügt.

Die hier gezeigte Vorgangsweise ist die einfachste Form, Komponentenbäume in Facelets aus mehreren Kompositionen aufzubauen. Wir werden Ihnen im Laufe der nächsten Abschnitte noch weitere Möglichkeiten zeigen, Seitendeklarationen modular aufzubauen.

4.2.2 Tag-Bibliotheken mit Facelets erstellen

Wir haben mittlerweile mit der Core-, der HTML- und der Facelets-Tag-Library drei verschiedene Tag-Bibliotheken kennengelernt. Jede von ihnen bietet unter einem im System eindeutigen Namensraum verschiedenste Tags zum einfachen Aufbau von Seitendeklarationen an. Wie wäre es, wenn Sie für Ihre eigenen Komponenten, Konverter und Validatoren auch eigene Tags definieren könnten? Das würde die tägliche Arbeit mit JSF doch erheblich vereinfachen. Facelets bietet auch dafür eine einfache Lösung an.

Eine benutzerdefinierte Tag-Bibliothek erlaubt die Definition von Tags für eigene Komponenten, Konverter und Validatoren. Wie die Tag-Bibliotheken der Standardkomponenten hat auch jede benutzerdefinierte Tag-Bibliothek einen im System eindeutigen Namensraum, mit dem sie in jede Seitendeklaration eingebunden werden kann. Neben Tag-Definitionen kann eine Tag-Bibliothek auch sogenannte EL-Funktionen enthalten, mit denen statische Funktionen in EL-Ausdrücken verfügbar gemacht werden.

Nachdem wir bis jetzt noch keine eigenen Java-basierten Komponenten erstellt haben, werden wir mit der Definition eines entsprechenden Tags noch bis Kapitel 5 warten. Wir wollen den folgenden Abschnitt mit der Definition einer EL-Funktion beginnen, wobei wir Ihnen auch gleich zeigen, wie Sie eine Tag-Bibliothek erstellen und im System registrieren können. Was wir Ihnen überdies nicht vorenthalten wollen, ist das Erstellen eines Tags für einen Konverter und einen Validator.

Definition einer EL-Funktion

JavaServer Pages ab Version 2.1 und Facelets bieten die Möglichkeit, statische Funktionen in EL-Ausdrücken verfügbar zu machen – und das mit einer beliebigen Anzahl von Parametern. Nachdem die Beispiele im Buch Facelets als Seitendeklarationsprache einsetzen, werden wir uns an dieser Stelle auf die Definition einer EL-Funktion mit Facelets beschränken. In JSP funktioniert die Definition allerdings sehr ähnlich.

Als Beispiel implementieren wir eine Funktion, die für ein Geburtsdatum das Alter berechnet und als Zahl zurückliefert. Der dazu notwendige Java-Code beschränkt sich auf wenige Zeilen in der statischen Methode `getAge` der Klasse `MyGourmetUtil`. Diese Klasse ist in Listing 4.6 zu sehen.

Listing 4.6
Java-Code der EL-Funktion

```

1 public class MyGourmetUtil {
2     public static int getAge(Date birthday) {
3         Calendar birthCal = Calendar.getInstance();
4         birthCal.setTime(birthday);
5         Calendar today = Calendar.getInstance();
6         int age = today.get(Calendar.YEAR)
7             - birthCal.get(Calendar.YEAR);
8         if (today.get(Calendar.DAY_OF_YEAR)
9             < birthCal.get(Calendar.DAY_OF_YEAR))
10            age--;
11        return age;
12    }
13 }
```

Die hinter der EL-Funktion liegende Methode ist damit vorhanden, jetzt muss sie noch in einer Tag-Bibliothek verfügbar gemacht werden. Listing 4.7 zeigt die Tag-Bibliothek `mygourmet.taglib.xml` mit der Definition der EL-Funktion in einem `function`-Element.

Listing 4.7
Tag-Bibliothek mit einer EL-Funktion

```

1 <facelet-taglib version="2.0"
2     xmlns="http://java.sun.com/xml/ns/javaee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5         http://java.sun.com/xml/ns/javaee/
6         web-facelettaglibrary_2_0.xsd">
7     <namespace>http://at.irian/mygourmet</namespace>
8     <function>
9         <function-name>getAge</function-name>
10        <function-class>
11            at.irian.jsfatwork.gui.util.MyGourmetUtil
12        </function-class>
13        <function-signature>
14            int getAge(java.util.Date)
15        </function-signature>
16    </function>
17 </facelet-taglib>
```

Der Name, unter dem die Funktion später in EL-Ausdrücken einsetzbar ist, wird im Kindelement `function-name` angegeben. Die Klasse und die aufzurufende Methode werden in den Elementen `function-class` und

function-signature definiert. Sie müssen in beiden Werten qualifizierete Namen verwenden, damit die jeweiligen Klassen gefunden werden.

Das Einbinden der Tag-Bibliothek in die Anwendung erfolgt mit dem Kontextparameter `javax.faces.FACELETS_LIBRARIES` in der `web.xml`. Facelets interpretiert den Wert dieses Parameters als eine über Semikolons separierte Liste von Tag-Bibliotheken. Nach der Registrierung ist die Tag-Bibliothek über die im Element namespace definierte URI `http://at.irian/mygourmet` im System verfügbar. Listing 4.8 zeigt den Ausschnitt der `web.xml`-Datei.

```
<context-param>
  <param-name>
    javax.faces.FACELETS_LIBRARIES
  </param-name>
  <param-value>
    /WEB-INF/mygourmet.taglib.xml
  </param-value>
</context-param>
```

Listing 4.8

*Tag-Bibliothek in der
web.xml einbinden*

Facelets bindet Tag-Bibliotheken aus JAR-Dateien im Classpath automatisch ein, wenn sie im META-INF-Verzeichnis liegen und ihr Dateiname mit `.taglib.xml` endet.

Dem Einsatz der EL-Funktion steht jetzt nichts mehr im Weg. Die benutzerdefinierte Tag-Bibliothek `mygourmet.taglib.xml` wird ähnlich den bestehenden Tag-Bibliotheken eingebunden. In Listing 4.9 sehen wir die Ausgabe des Alters unter Zuhilfenahme unserer Funktion. Bitte beachten Sie, dass beim Aufruf der Funktion das Präfix `mg:` mit angegeben werden muss.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2   xmlns:h="http://java.sun.com/jsf/html"
3   xmlns:mg="http://at.irian/mygourmet">
4   ...
5   <h:outputText value=
6     "#{mg:getAge(customerBean.customer.birthday)}"/>
7   ...
8 </html>
```

Listing 4.9

EL-Funktion im Einsatz

Definition eines Konverter-Tags

In Abschnitt 2.11.2 haben wir bereits einen Konverter für die Postleitzahl definiert und unter dem Bezeichner `at.irian.ZipCode` registriert. Eingebunden wurde dieser Konverter mit `f:converter` unter Angabe

dieses Bezeichners. Schön wäre es, wenn ein eigenes Tag mit einem sprechenden Namen und der Möglichkeit, Attribute an diesen Konverter zu übergeben, existieren würde – nichts einfacher als das.

Das Hinzufügen der Zeilen in Listing 4.10 zu unserer Bibliothek reicht aus, um den Konverter unter dem Tag `convertZipCode` zur Verfügung zu stellen. Beim Aufbau des Komponentenbaums fügt Facelets dann für jedes Tag mit dem Namen `convertZipCode` aus unserer Bibliothek den Konverter mit dem Bezeichner `at.irian.ZipCode` ein.

Listing 4.10
Definition eines
Konverter-Tags

```
<tag>
  <tag-name>convertZipCode</tag-name>
  <converter>
    <converter-id>at.irian.ZipCode</converter-id>
  </converter>
</tag>
```

Listing 4.11 zeigt das neue Tag in einer Seitendeklaration. Als Voraussetzung gilt auch hier, dass die Tag-Bibliothek in der Deklaration unter dem Präfix `mg` bekannt gemacht wurde.

Listing 4.11
Einsatz des
benutzerdefinierten
Konverter-Tags

```
<h:inputText id="zipCode" size="30"
  value="#{addressBean.address.zipCode}">
  <mg:convertZipCode/>
</h:inputText>
```

In *MyGourmet 12* (Abschnitt 4.4.3) erstellen wir einen weiteren benutzerdefinierten Konverter mit eigenem Tag zum Umwandeln von *Collections* in Zeichenketten. Im nächsten Abschnitt über Validatoren zeigen wir Ihnen, wie auch Attribute übergeben werden können.

Definition eines Validator-Tags

Der Vorgang der Definition eines Tags für einen Konverter funktioniert in exakt der gleichen Weise auch für Validatoren. Obwohl in *MyGourmet* die Validierung über Bean-Validation abgewickelt wird, werden wir hier einen Validator für das Alter einer Person registrieren. Der Validator soll über die beiden optionalen Eigenschaften `minAge` und `maxAge` steuerbar sein.

Listing 4.12 zeigt die Zeilen für die Definition des Validator-Tags. Der interessante Aspekt an diesem Validator sind die beiden Eigenschaften `minAge` und `maxAge`.

Die Werte der beiden Eigenschaften können direkt über Attribute des Tags an den Validator übergeben werden. Facelets verknüpft diese dann automatisch mit gleichnamigen Eigenschaften der dahin-

```
<tag>
  <tag-name>validateAge</tag-name>
  <validator>
    <validator-id>at.irian.Age</validator-id>
  </validator>
</tag>
```

Listing 4.12
Definition eines
Validator-Tags

terliegenden Validator-Objekte. In Listing 4.13 sehen Sie das Tag `mg:validateAge` mit gesetztem Attribut `minAge` im Einsatz.

```
<h:inputText id="birthday" size="30"
  value="#{customerBean.customer.birthday}">
  <f:convertDateTime pattern="dd.MM.yyyy"/>
  <mg:validateAge minAge="18"/>
</h:inputText>
```

Listing 4.13
Einsatz des
benutzerdefinierten
Validator-Tags

4.2.3 MyGourmet 10: Advanced Facelets

Das Beispiel *MyGourmet 10* fasst alle Änderungen aus Abschnitt 4.2 zusammen. Ein Großteil der Neuerungen hat direkt oder indirekt mit der neuen Tag-Bibliothek `/WEB-INF/mygourmet.taglib.xml` zu tun, die unter dem Namensraum `http://at.irian/mygourmet` in der Anwendung verfügbar ist.

Alle Ansichten haben jetzt einen einheitlichen Seitenkopf, der über `mg:pageHeader` eingebunden ist. Genauso gut wäre es möglich, direkt das dahinterliegende Seitenfragment `header.xhtml` aus dem Verzeichnis `/WEB-INF/includes` über `ui:include` zu verwenden.

Der Konverter für die Postleitzahl in `editAddress.xhtml` und der Validator für das Alter des Kunden in `editCustomer.xhtml` sind jetzt direkt über Tags aus der neuen Bibliothek eingebunden. In der Ansicht `showCustomer.xhtml` wird zusätzlich das Alter der Person über die EL-Funktion `mg:getAge` ausgegeben.

4.3 Templating

Layout und Design spielen bei der Entwicklung vieler Webanwendungen eine wichtige Rolle. Neben einem ausgefeilten grafischen Design ist eine konsistente und durchgängige Seitenstruktur oft die Grundvoraussetzung für den Erfolg einer Applikation. Ein einheitliches Seitenlayout vereinfacht nicht nur die Bedienbarkeit für den Benutzer, sondern ermöglicht auch die konsequente Umsetzung eines Unternehmensdesigns

(Corporate Identity) auf allen Seiten. Diese Anforderungen lassen sich in der Entwicklung mithilfe von Templates umsetzen.

Der Einsatz von Templates verringert nicht nur die Redundanz der erstellten Anwendung, sondern bietet auch entscheidende Vorteile während der Entwicklung. Templates fördern durch den modularen Aufbau der Seiten die Wiederverwendung von Code und erleichtern die Trennung von Design und Inhalt. Diese Entkopplung unterstützt eine konsequente Durchsetzung des Designs im gesamten Projekt und schwächt die Auswirkung von nachträglichen Änderungen ab. Im Idealfall muss dann nur das Template oder ein zentral definiertes Seitenfragment angepasst werden, was Entwicklungs- und Wartungskosten spart.

Facelets bietet eine sehr elegante Templating-Lösung, die perfekt in den JSF-Lebenszyklus integriert ist. Ein Template ist in Facelets in erster Linie eine XHTML-Datei – wie jede andere Seitendeklaration. Den Unterschied macht das Tag `ui:insert` aus der *Facelets-Tag-Library*, mit dem ersetzbare Bereiche im Template definiert werden können. Eine Seitendeklaration, die auf diesem Template aufbaut (der sogenannte Template-Client), kann diese Bereiche mit dem eigentlichen Content ersetzen. Die komplette Ansicht besteht dann aus dem im Template definierten Inhalt und den ersetzten Bereichen aus dem Template-Client.

Sehen wir uns nun anhand eines kleinen Beispiels an, wie das Templating mit Facelets in der Praxis aussieht. Die Seiten dieses Beispiels sollen über eine Kopfzeile, einen Content-Bereich und eine Fußzeile verfügen. Diese Anforderung setzen wir in Form eines Templates mit der entsprechenden Struktur und drei ersetzbaren Bereichen um. Dadurch ist das Layout zentral definiert und einfach auf alle Seiten anwendbar. Das entsprechende Template mit dem Namen `template.xhtml` ist in Listing 4.14 zu finden.

Das Template ist ein einfaches XHTML-Dokument, in dem die grundlegende Seitenstruktur mit `div`-Elementen abgebildet ist. Die drei `ui:insert`-Tags mit den Namen `header`, `content` und `footer` definieren die ersetzbaren Bereiche. Bei den `ui:insert`-Bereichen für die Kopf- und die Fußzeile nutzen wir die Möglichkeit, Default-Content zu definieren. Falls ein Template-Client den entsprechenden Bereich nicht überschreibt, fügt Facelets den Inhalt innerhalb des `ui:insert`-Tags in die Ausgabe ein. Diese Vorgehensweise ist besonders dann praktikabel, wenn der Inhalt in weiten Teilen der Applikation gleich bleibt.

Im nächsten Schritt werden wir die Seite `showCustomer.xhtml` erstellen. Sie basiert auf unserem Template und definiert einen eigenen Content-Bereich. Facelets bietet dafür die Tags `ui:composition` und `ui:define` an. `ui:composition` stellt eine Verbindung zum Template mit dem im Attribut `template` angegebenen Namen her – in unserem Fall `template.xhtml`. Innerhalb von `ui:composition` können die

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:h="http://java.sun.com/jsf/html"
3     xmlns:ui="http://java.sun.com/jsf/facelets">
4 <head>
5   <title>MyGourmet</title>
6   <link rel="stylesheet" type="text/css" href="style.css"/>
7 </head>
8 <body>
9   <div id="header">
10    <ui:insert name="header">
11      <h1>MyGourmet</h1>
12    </ui:insert>
13  </div>
14  <div id="content">
15    <ui:insert name="content"/>
16  </div>
17  <div id="footer">
18    <ui:insert name="footer">
19      <h:outputText value="Copyright (c) 2009"/>
20    </ui:insert>
21  </div>
22 </body>
23 </html>
```

Listing 4.14

Beispiel eines
Templates in Facelets

Zielbereiche des Templates mit `ui:define`-Blöcken überschrieben werden. Welcher mit `ui:insert` definierte Bereich des Templates durch den `ui:define`-Block ersetzt wird, bestimmt das Attribut `name`.

Bevor wir etwas genauer analysieren, wie Facelets eine Ansicht mit einem Template rendert, werfen wir in Listing 4.15 noch einen Blick auf den kompletten Template-Client `showCustomer.xhtml`.

Wie baut Facelets die Ansicht aus `showCustomer.xhtml` mit dem Template auf? Wie schon beim Einsatz von `ui:include` ignoriert Facelets auch hier sämtliche Inhalte außerhalb `ui:composition` und der Aufbau des Komponentenbaums beginnt mit dem referenzierten Template. Während des Seitenerstellungsvorgangs werden die mit `ui:insert` definierten Bereiche im Template ersetzt. In unserem Beispiel kommt der Inhalt der Kopf- und Fußzeile aus dem Template und der Inhalt des Content-Bereichs stammt aus dem `ui:define`-Block in `showCustomer.xhtml`.

Abbildung 4.1 zeigt die ersetzbaren `ui:insert`-Bereiche des Templates anhand der gerenderten Ausgabe unseres Beispiels. Die Umrahmungen mit den Namen der einzelnen Teile in der linken oberen Ecke dienen nur der besseren Visualisierung und wurden nicht von JSF gerendert.

Listing 4.15

Beispiel eines
Template-Clients in
Facelets

```

1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:h="http://java.sun.com/jsf/html"
3     xmlns:ui="http://java.sun.com/jsf/facelets">
4 <head>
5   <title>Show Customer</title>
6 </head>
7 <body>
8   <ui:composition template="template.xhtml">
9     <ui:define name="content">
10      <h2>Kundendaten</h2>
11      <h:panelGrid id="grid" columns="2">
12        <h:outputText value="Vorname:" />
13        <h:outputText value="#{customer.firstName}" />
14        <h:outputText value="Nachname:" />
15        <h:outputText value="#{customer.lastName}" />
16      </h:panelGrid>
17    </ui:define>
18  </ui:composition>
19 </body>
20 </html>

```

Abbildung 4.1

Ersetzbare Bereiche des
Templating-Beispiels



Facelets bietet für das Templating noch einiges mehr als die im letzten Abschnitt beschriebene Basisfunktionalität. Nach der Vorstellung von mehrstufigem Templating in Abschnitt 4.3.1 werden wir in Abschnitt 4.3.2 einen Blick auf den Einsatz von mehreren Templates in einem Template-Client werfen.

4.3.1 Mehrstufiges Templating

Mehrstufiges Templating ermöglicht den Aufbau einer Hierarchie von Templates. Das ist besonders dann praktisch, wenn eine Anwendung in mehrere Bereiche gegliedert ist, die ein gemeinsames Layout, aber un-

terschiedliche Inhalte haben. Im Fall von *MyGourmet* wäre das zum Beispiel ein Kundenbereich zum Bestellen von Gerichten, ein Bereich für Restaurants und Anbieter und ein allgemeiner Administrationsbereich. Das grundlegende Layout der Seiten mit Kopfzeile, linker Seitenleiste, Content-Bereich und Fußzeile bleibt gleich und wird daher im Haupttemplate aufgebaut. Dort landet auch ein Standardwert für den Inhalt der Kopf- und Fußzeile. In den abgeleiteten Templates wird die linke Seitenleiste überschrieben und mit bereichsspezifischem Inhalt gefüllt. Der Content-Bereich bleibt weiterhin leer und wird erst in den konkreten Seiten überschrieben. Abbildung 4.2 zeigt die mehrstufige Templating-Hierarchie in *MyGourmet* inklusive der bereits bekannten Seite `showCustomer.xhtml` aus dem Kundenbereich.

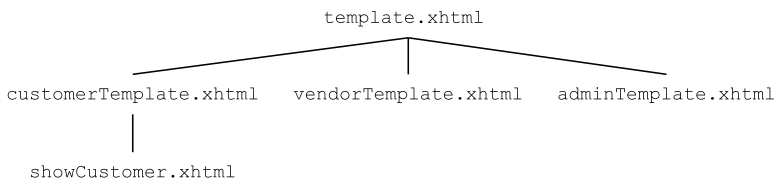


Abbildung 4.2
Templating-Hierarchie
von *MyGourmet*

Der Aufbau einer mehrstufigen Templating-Hierarchie gestaltet sich einfach, da jeder Template-Client wiederum die Rolle eines Templates einnehmen kann – in Facelets gibt es keine strikte Trennung zwischen diesen beiden Rollen. Die oben erwähnten und in Abbildung 4.2 ersichtlichen Templates für die Anwendungsbereiche nehmen beide Rollen ein. Einerseits sind sie Template-Clients, da sie mit folgendem Code das Haupttemplate referenzieren:

```
<ui:composition template="template.xhtml">
```

Für die konkreten Seiten im Anwendungsbereich sind sie allerdings Templates, die neben den geerbten Inhalten aus dem Haupttemplate den Inhalt der linken Seitenleiste deklarieren. In der Seite `customerPage.xhtml` wird das Template dann beispielsweise mit folgendem Code referenziert:

```
<ui:composition template="customerTemplate.xhtml">
```

Eine genauere Betrachtung des mehrstufigen Templatings in der Praxis folgt mit Beispiel *MyGourmet 11* in Abschnitt 4.3.3.

4.3.2 Mehrere Templates pro Seite

In manchen Fällen macht es Sinn, neben einem Template für die Ansicht selbst zusätzliche Templates für wiederkehrende Bereiche der Seite zu verwenden. Denken Sie zum Beispiel an speziell gestaltete Bereiche in

einer Seitenleiste oder an Vorlagen für unterschiedliche Typen von Seiteninhalten. Die bereits bekannte Methode mit `ui:composition` führt in diesem Fall nicht zum Erfolg, da der Content außerhalb des Tags abgeschnitten wird. Bei zwei geschachtelten `ui:composition`-Tags mit gesetztem `template`-Attribut gewinnt immer das innere – dieser Ansatz ist also für unsere Zwecke nicht brauchbar.

Facelets bietet aber auch dafür eine Lösung an. Mit `ui:decorate` existiert eine Variante von `ui:composition`, bei deren Verwendung der außerhalb des Tags liegende Code nicht abgeschnitten wird. Wie der Name bereits sagt, wird der Content innerhalb von `ui:decorate` mit dem Inhalt des referenzierten Templates dekoriert.

Sehen wir uns das anhand eines kleinen Beispiels an. Listing 4.16 zeigt einen Ausschnitt aus dem Quelltext eines Template-Clients, der ein Template für die Seite und eines für eine Box in der Seitenleiste beinhaltet.

Listing 4.16
Template-Client mit
mehreren Templates

```
1 <ui:composition template="template.xhtml">
2   ...
3   <ui:define name="left_sidebar">
4     <ui:decorate template="sideBox.xhtml">
5       <ui:param name="title" value="Meldungen"/>
6       <h:outputText value="#{bean.msg}"/>
7     </ui:decorate>
8   </ui:define>
9   ...
10 </ui:composition>
```

Innerhalb von `ui:decorate` wird dem referenzierten Template `sideBox.xhtml` mit dem Tag `ui:param` der Parameter mit dem Namen `title` übergeben. Der restliche Inhalt von `ui:decorate` bildet den Inhalt der Box.

Das Template für die Box ist ein XHTML-Dokument mit einer kleinen Besonderheit. Da es sich um ein Template für einen Teil der kompletten Seite handelt, darf das HTML-Grundgerüst nicht gerendert werden. Dazu dient das Tag `ui:composition` – diesmal allerdings ohne das Attribut `template`. Auf diese Weise eingesetzt definiert es einen Teilkomponentenbaum bestehend aus seinem Inhalt. Alle Elemente außerhalb werden abgeschnitten.

Der im Template-Client gesetzte Parameter `title` ist im Template als Variable verfügbar und wird über einen EL-Ausdruck referenziert. Der Inhalt der Box steht direkt im `ui:decorate`-Tag. Durch den Einsatz von `ui:insert` ohne das Attribut `name` fügt Facelets beim Rendern den kompletten Inhalt von `ui:decorate` ein. Das komplette Template für die Box finden Sie in Listing 4.17.

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:f="http://java.sun.com/jsf/core"
3     xmlns:h="http://java.sun.com/jsf/html"
4     xmlns:ui="http://java.sun.com/jsf/facelets">
5 <head>
6   <title>Template for a box in a side bar</title>
7 </head>
8 <body>
9   <ui:composition>
10    <div class="side_box">
11      <p class="header">#{title}</p>
12      <ui:insert>Default body</ui:insert>
13    </div>
14  </ui:composition>
15 </body>
16 </html>
```

Listing 4.17
*Template für eine
Seitenleiste*

Das Beispiel *MyGourmet 11* in Abschnitt 4.3.3 benutzt ebenfalls mehrere Templates pro Seite, um Boxen in der Seitenleiste zu formatieren.

4.3.3 *MyGourmet 11*: Templating mit Facelets

Nach der Vorstellung der Templating-Fähigkeiten von Facelets wird es Zeit, diese in unserem Beispiel anzuwenden. *MyGourmet 11* erweitert das Vorgängerbeispiel *MyGourmet 10* um ein einfaches, mithilfe von Templates umgesetztes Layout. Bevor wir allerdings auf die Details der Implementierung eingehen, wollen wir kurz präsentieren, wie die Anwendung im Browser aussieht. Abbildung 4.3 zeigt die gerenderte Ausgabe der Seite *showCustomer.xhtml*.

In diesem Beispiel kommt die bereits kurz vorgestellte mehrstufige Template-Hierarchie aus Abbildung 4.2 zum Einsatz. Das Haupttemplate mit dem Namen *template.xhtml* bietet nicht viel Neues. Es definiert das grundlegende Layout der Anwendung mit je einem *div*-Container für die Kopfzeile, die linke Seitenleiste, den Content-Bereich und die Fußzeile. Innerhalb dieser Container befindet sich je ein *ui:insert*-Tag mit einem eindeutigen Namen. Das in Abbildung 4.3 ersichtliche Design ist in einem verlinkten CSS-Dokument definiert.

Wenden wir uns nun dem Template für die Seiten im Kundenbereich (*customerTemplate.xhtml*) zu. Es leitet sich vom Haupttemplate ab und definiert die Standardinhalte der Kopf- und Fußzeile und der Seitenleiste für den Kundenbereich von *MyGourmet*. Einen Ausschnitt zeigt Listing 4.18.

Die Definition der Kopf- und Fußzeile erfolgt direkt im Template – die Inhalte sind relativ überschaubar. Für die Seitenleiste kommt

Abbildung 4.3
MyGourmet 11:
Kundenseite im
Browser



Listing 4.18
MyGourmet 11:
Template für
Kundenseiten

```

1 <ui:composition template="template.xhtml">
2   <ui:define name="header">
3     <h:graphicImage value="/images/logo.png"/>
4     <h1>#{msgs.title_main}</h1>
5   </ui:define>
6   <ui:define name="left_sidebar">
7     <ui:include src="leftSideBar.xhtml"/>
8   </ui:define>
9   <ui:define name="footer">
10    <h:outputText value="#{msgs.footer_left_text}"
11      style="float: left;"/>
12    <h:outputText value="#{msgs.footer_right_text}"
13      style="float: right;"/>
14  </ui:define>
15 </ui:composition>

```

jedoch eine andere Vorgehensweise zum Einsatz: Ihr Inhalt wird im separaten XHTML-Dokument `leftSideBar.xhtml` definiert und mit dem Tag `ui:include` in das Template eingebunden.

Die Seitenleiste besteht aus einer Box mit einem kleinen Menü und einer Box mit aktuellen Meldungen. Die Umsetzung entspricht dem in Abschnitt 4.3.2 vorgestellten Beispiel für den Einsatz von mehreren Templates mit `ui:decorate`. Listing 4.19 zeigt einen Ausschnitt der `leftSideBar.xhtml`-Datei.

```
1 <ui:composition>
2   <ui:decorate template="/META-INF/templates/sideBox.xhtml">
3     <ui:param name="title" value="#{msgs.menu_title}"/>
4     <h:form id="menu">
5       <h:panelGrid columns="1">
6         <h:commandLink action="showCustomer">
7           #{msgs.menu_show_customer}
8         </h:commandLink>
9       </h:panelGrid>
10    </h:form>
11  </ui:decorate>
12  <ui:decorate template="/META-INF/templates/sideBox.xhtml">
13    <ui:param name="title" value="#{msgs.news_title}"/>
14    <p>MyGourmet - jetzt mit Facelets und Templating</p>
15  </ui:decorate>
16 </ui:composition>
```

Listing 4.19

*MyGourmet 11: Linke
Seitenleiste*

4.4 Bookmarks und GET-Anfragen in JSF

JSF bis Version 1.2 ist nur eingeschränkt in der Lage, Bookmarks zu setzen und mit GET-Anfragen umzugehen. Das liegt vor allem an der Tatsache, dass jeder Klick auf eine `h:commandLink`- oder `h:commandButton`-Komponente eine POST-Anfrage auslöst. Diesem Umstand wird in Version 2.0 der Spezifikation mit einer erweiterten Unterstützung von GET-Anfragen Rechnung getragen.

4.4.1 Navigation mit `h:link` und `h:button`

Die Basis der GET-Unterstützung in JSF 2.0 bilden mit `h:link` und `h:button` zwei neue Komponenten, die als Link beziehungsweise Schaltfläche gerendert werden und eine GET-Anfrage absetzen. Der Clou ist, dass dabei trotzdem der Navigationsmechanismus von JSF zum Einsatz kommt. Zu diesem Zweck haben beide Komponenten das Attribut `outcome`, dessen Wert zum Auflösen der URL an den Navigation-Handler übergeben wird. Dieser versucht zuerst einen Navigationsfall zu finden, für den `from-outcome` mit dem Wert von `outcome` übereinstimmt. Bleibt die Suche erfolglos, wird der Wert von `outcome` direkt als View-ID interpretiert. Im Unterschied zur klassischen Navigation wird die View-ID bereits beim Rendern der Ansicht aufgelöst und nicht dynamisch in der Invoke-Application-Phase beim Postback. Dieses Konzept wird daher auch als *präemptive Navigation* bezeichnet.

Zur Demonstration der neuen GET-Fähigkeiten von JSF 2.0 erhält *MyGourmet* zwei neue Ansichten. Die erste neue Seite mit der View-ID

providerList.xhtml zeigt eine Liste von Anbietern, die Essen ausliefern. Jeder Eintrag dieser Liste verweist mit einem `h:link`-Tag auf die Detailseite `showProvider.xhtml`. Listing 4.20 zeigt einen Ausschnitt der Seitendeklaration `providerList.xhtml` mit dem Link zur Detailseite.

Listing 4.20
h:link im Einsatz

```

1 <h:dataTable var="provider"
2   value="#{providerBean.providerList}">
3   <h:column>
4     <f:facet name="header">
5       <h:outputText value="#{msgs.provider_name}"/>
6     </f:facet>
7     <h:link outcome="showProvider"
8       value="#{provider.name}">
9       <f:param name="id" value="#{provider.id}"/>
10    </h:link>
11  </h:column>
12 </h:dataTable>

```

Der Wert des Attributs `value` wird dabei als Linktext gerendert. Wir nutzen an dieser Stelle die implizite Navigation und geben im Attribut `outcome` direkt die View-ID der Detailseite an. Der eindeutige Bezeichner des entsprechenden Anbieters wird mit dem Tag `f:param` als Kind der Linkkomponente bereitgestellt.

Listing 4.21 zeigt, wie JSF 2.0 die Links rendert. Hier sehen Sie auch, was der Begriff *präemptive Navigation* in der Praxis bedeutet. Bereits beim Rendern der Ansicht wird für jede `h:link`- oder `h:button`-Komponente die resultierende URL abhängig vom Attribut `outcome` bestimmt. Aktiviert der Benutzer einen der Links beziehungsweise eine der Schaltflächen, schickt der Browser eine simple GET-Anfrage an den Server und es gibt in diesem Fall keinen Postback. Deswegen ist es auch nicht notwendig, `h:link` und `h:button` in ein `h:form`-Tag einzubetten.

Listing 4.21
Gerenderte Ausgabe
von h:link

```

<a href="/showProvider.jsf?id=1">Pizzeria Venezia</a>
<a href="/showProvider.jsf?id=2">Rhodos</a>
<a href="/showProvider.jsf?id=3">Frying Dutchman</a>

```

Nachdem der Browser eine GET-Anfrage schickt, stimmt auch die URL in der Adressleiste mit der tatsächlich gerenderten Seite überein. Der Anwender kann daher auch ein Lesezeichen auf diese Ansicht setzen. Das hört sich trivial an, trifft aber bei der klassischen Navigation nicht immer zu, da die Steuerkomponenten `h:commandLink` und `h:commandButton` erst einen Postback auf die aktuelle Seite machen, bevor JSF die Navigation ausführt und eine neue Ansicht rendert. Daher hinkt die Adressleiste um eine Ansicht hinterher.

4.4.2 View-Parameter

Wenn der Benutzer den von `h:link` gerenderten Link aktiviert, muss JSF den übergebenen Parameter `id` verarbeiten und den richtigen Anbieter anzeigen. Dabei kommen die sogenannten View-Parameter ins Spiel, die Request-Parameter direkt ans Modell binden. Diese Parameter sind im Grunde nichts anderes als Eingabekomponenten, die über Request-Parameter befüllt werden. Wie bei herkömmlichen Eingabekomponenten werden auch hier die Werte zuerst konvertiert und anschließend validiert.

View-Parameter werden in einer Seitendeklaration innerhalb des Tags `f:metadata` in Form von `f:viewParam`-Komponenten angegeben. Listing 4.22 zeigt den `f:metadata`-Bereich der Ansicht `showProvider.xhtml`. Der darin eingebettete View-Parameter verbindet den Request-Parameter mit dem Namen `id` direkt mit der Eigenschaft `providerBean.id` in der Backing-Bean.

```
<f:metadata>
  <f:viewParam name="id" value="#{providerBean.id}"/>
</f:metadata>
```

Listing 4.22
View-Parameter im Einsatz

Sie können überprüfen, ob sich der View-Parameter wie eine Eingabekomponente verhält (oder treffender gesagt eine ist): Rufen Sie die Seite im Browser mit einem nichtnumerischen Wert für den Parameter `id` auf. Das Ergebnis ist eine Fehlermeldung des JSF-Number-Konverters, da die Eigenschaft `providerBean.id` den Typ `long` aufweist. Es ist auch ohne Weiteres möglich, einen Validator einzusetzen oder die Eigenschaft mit Bean-Validation-Metadaten zu versehen.

Nachdem der Bezeichner erfolgreich in die Bean übertragen wurde, müssen vor dem Rendern der Ansicht noch die Daten des Anbieters geladen werden. JSF 2.0 gibt uns mit System-Events das richtige Werkzeug dafür in die Hand. Wir verschieben die Erläuterung der Details in den Abschnitt 4.4.3 über *MyGourmet 12*.

Wenn View-Parameter auf einer Seite angegeben sind, können diese automatisch in `h:link`- und `h:button`-Elemente übernommen werden, indem deren Attribut `includeViewParams` auf `true` gesetzt wird. Als Beispiel fügen wir auf `showProvider.xhtml` den in Listing 4.23 gezeigten Link auf eine Seite zum Bearbeiten des Anbieters hinzu. Der Wert des View-Parameters in der Ansicht `showProvider.xhtml` wird automatisch als Parameter im Link verwendet, wodurch sich in der HTML-Ausgabe des Links folgende URL ergibt: `/editProvider.xhtml?id=1`.

Listing 4.23

*h:link mit
View-Parametern*

```
<h:link outcome="editProvider"
      includeViewParams="true"
      value="#{msgs.edit_provider}"/>
```

Positionierung von f:metadata

Das Tag `f:metadata` muss immer ein direktes Kind von `f:view` sein und darf nicht in ein Template oder ein mit `ui:include` eingefügtes Seitenfragment ausgelagert werden. Die Verbindung von Templating und View-Parametern lässt sich dennoch sehr einfach bewerkstelligen. Dazu muss das Template einen ersetzbaren Bereich für die View-Parameter definieren, der dann im Template-Client ersetzt wird. Listing 4.24 zeigt ein Beispiel für ein Template mit dem ersetzbaren Bereich `metadata`.

Listing 4.24

*Template mit
View-Parametern*

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:ui="http://java.sun.com/jsf/facelets"
3     xmlns:f="http://java.sun.com/jsf/core">
4 <body>
5   <f:view>
6     <ui:insert name="metadata"/>
7     <div id="content">
8       <ui:insert name="content"/>
9     </div>
10  </f:view>
11 </body>
12 </html>
```

Listing 4.25 zeigt einen Template-Client, der auf dem zuvor definierten Template aufbaut und den View-Parameter-Bereich überschreibt. Wie Sie sehen, muss das komplette `f:metadata`-Tag im Template-Client deklariert werden.

Lebenszyklus mit View-Parametern

Wir klären jetzt noch die Frage, wie sich die Ausführung des Lebenszyklus ändert, wenn View-Parameter ins Spiel kommen. Bei einer initialen Anfrage auf eine Ansicht handelt es sich ja immer um eine GET-Anfrage. Vor JSF 2.0 sprang bei einer solchen Anfrage die Ausführung des Lebenszyklus nach Phase 1 sofort zu Phase 6 und die Ansicht wurde gerendert.

Dieses Verhalten hat sich verändert: In Phase 1 wird zunächst geprüft, ob es einen Metadatenbereich und View-Parameter gibt. Wenn ja, wird eine neue Ansicht erzeugt, die nur die View-Parameter enthält. Damit wird dann der komplette Lebenszyklus durchlaufen. Gibt

```
1 <html xmlns="http://www.w3.org/1999/xhtml"
2     xmlns:ui="http://java.sun.com/jsf/facelets"
3     xmlns:f="http://java.sun.com/jsf/core">
4 <body>
5   <ui:composition template="template.xhtml">
6     <ui:define name="metadata">
7       <f:metadata>
8         <f:viewParam name="id" value="bean.id"/>
9       </f:metadata>
10    </ui:define>
11    <ui:define name="metadata">
12      Page content
13    </ui:define>
14  </ui:composition>
15 </body>
16 </html>
```

Listing 4.25
*Template-Client mit
View-Parametern*

es in der Anfrage Parameter, werden die Daten in den folgenden Phasen wie bei einem Postback behandelt: Zuerst werden Sie den einzelnen View-Parametern zugeordnet, dann konvertiert und validiert und ins Modell zurückgeschrieben, falls keine Fehler aufgetreten sind. Abschließend wird die Ansicht wie bisher gerendert.

4.4.3 *MyGourmet 12*: View-Parameter und GET

Das Beispiel *MyGourmet 12* fasst alle Änderungen rund um das Thema View-Parameter zusammen. Die augenscheinlichste Neuerung ist der neue Anbieterbereich in der Anwendung. Als Einstiegspunkt dient die Seite `providerList.xhtml` mit einer Übersicht der Anbieter. Diese Liste wird beim Erzeugen der Backing-Bean `ProviderBean` mit einigen Werten initialisiert. Von dieser Seite führt pro Anbieter eine `h:link`-Komponente zu `showProvider.xhtml`. Die Daten eines Anbieters werden in Instanzen der Klasse `Provider` abgelegt.

Wir wollen nun unser weiter oben gegebenes Versprechen einlösen und auf das Laden der Anbieterdaten mithilfe von System-Events eingehen. System-Events werden zu bestimmten Zeitpunkten im Lebenszyklus ausgelöst. JSF 2.0 definiert eine ganze Reihe dieser Ereignisse, für die Listener registriert werden können. Uns interessiert hier das `PreRenderViewEvent`. Dieses Ereignis wird kurz vor dem Rendern der Ansicht ausgelöst und erfüllt somit genau unsere Anforderungen.

Über das Tag `f:event` registrieren wir direkt in der Seitendeklaration die Methode `preRenderView()` der Backing-Bean als Listener für dieses Ereignis. Hier die Registrierung in `showProvider.xhtml`:

```
<f:event type="javax.faces.event.PreRenderViewEvent"
  listener="#{providerBean.preRenderView}"/>
```

Listing 4.26 zeigt die zuvor registrierte Listener-Methode in der Klasse `ProviderBean`. Hier erfolgt das Laden der Anbieterdaten, falls beim Konvertieren und Validieren des View-Parameters kein Fehler aufgetreten ist. Seit JSF 2.0 lässt sich diese Abfrage einfach über die Methode `isValidationFailed` am Faces-Context bewerkstelligen. Sollte kein Anbieter mit dem angegebenen Bezeichner existieren, erzeugen wir eine entsprechende Nachricht. Damit endet der kurze Ausflug zum Thema System-Events.

Listing 4.26
MyGourmet 12:
Listener-Methode für
ein System-Event

```
1 public void preRenderView(ComponentSystemEvent ev) {
2     FacesContext ctx = FacesContext.getCurrentInstance();
3     if (!ctx.isValidationFailed()) {
4         this.provider = findProvider(id);
5         if (provider == null) {
6             ctx.addMessage(null, GuiUtil.getFacesMessage(ctx,
7                 FacesMessage.SEVERITY_ERROR,
8                 "error_non_existing_provider", id));
9         }
10    }
11 }
```

Da wir seit *MyGourmet 11* Templating benutzen, mussten wir das Template (wie in Abschnitt 4.4.2 beschrieben) für den Einsatz von View-Parametern anpassen.

Eine weitere kleine Änderung betrifft die linke Seitenleiste. Damit der Anbieterbereich erreichbar ist, haben wir das Menü um einen Link erweitert. Zur besseren Unterstützung von Bookmarking haben wir das Menü auf `h:link`-Komponenten umgestellt. Als angenehmer Nebeneffekt kann dadurch die Form eingespart werden.

Listing 4.27 zeigt den Einsatz eines neuen Konverters, der *Collections* in Zeichenketten umwandelt. Mit dem Attribut `bundle` kann ihm ein Resource-Bundle übergeben werden, aus dem die Einträge der Liste aufgelöst werden. Nachdem mittlerweile der Anbieter und der Kunde eine Liste von Kategorien aufweisen, die aber nur symbolische Konstanten enthalten, spart dieser Konverter einiges an Quelltext.

Listing 4.27
MyGourmet 12: Einsatz
des Listenkonverters

```
<h:outputText value="#{providerBean.provider.categories}">
  <mg:convertList separator=", " bundle="#{msgs}"/>
</h:outputText>
```

Listing 4.28 zeigt die `getAsString`-Methode und die Eigenschaften der Konverterklasse `ListConverter`, die über die Attribute des Custom-

Tags `mg:convertList` gesetzt werden. Dieses Tag wurde, wie schon im letzten Beispiel gezeigt, in der Tag-Bibliothek `mygourmet.taglib.xml` definiert.

```
1 private String separator;
2 private ResourceBundle bundle;
3
4 public String getAsString(FacesContext ctx,
5     UIComponent comp, Object value) {
6     StringBuilder builder = new StringBuilder();
7     if (value instanceof Collection) {
8         for (Object obj : (Collection)value) {
9             String item = obj.toString();
10            if (builder.length() > 0 && separator != null) {
11                builder.append(separator);
12            }
13            if (bundle != null) {
14                builder.append(
15                    GuiUtil.getResourceText(bundle, item));
16            } else {
17                builder.append(item);
18            }
19        }
20    }
21    return builder.toString();
22 }
```

Listing 4.28

*MyGourmet 12:
Listenkonverter*

4.5 Verwaltung von Ressourcen

In JSF handelt es sich bei Ressourcen um Artefakte wie Bilder, Skripte oder *Stylesheets*, die eine Komponente benötigt, um vollständig am Client angezeigt zu werden.

Die Verwaltung von Ressourcen ist eine Anforderung, die in der einen oder anderen Form für jede Anwendung relevant ist. Besonders für Komponentenbibliotheken ist es wichtig, die Abhängigkeiten zwischen Komponenten und Ressourcen wie Skripten oder Stylesheets zu definieren. Außerdem müssen diese Ressourcen für den Anwendungsentwickler transparent zur Verfügung gestellt werden. Vor JSF 2.0 hat es dafür keinen standardisierten Weg gegeben. Viele Anbieter haben daher eigenständige Lösungen entwickelt, die allerdings in den wenigsten Fällen miteinander kompatibel sind.

Mit JSF 2.0 gibt es jetzt einen standardisierten Weg, Ressourcen zu verwalten und flexibel in der Ansicht zu positionieren. Davon profitie-

ren nicht nur Komponentenbibliotheken, sondern alle Anwendungen mit eigenen Bildern, Stylesheets oder Skripten.

4.5.1 Identifikation von Ressourcen – Teil 1

Ressourcen werden in JSF 2.0 im einfachsten Fall durch ihren Namen identifiziert. Sehen wir uns das am besten anhand eines Beispiels an. Folgender Code fügt das Bild `image.png` in eine Ansicht ein:

```
<h:graphicImage name="image.png"/>
```

Erfahrene JSF-Entwickler werden sofort bemerkt haben, dass in JSF 1.2 noch kein Attribut `name` für das `h:graphicImage`-Tag definiert ist. Der Wert dieses Attributs ist der Name der Ressource, die als Bild ausgegeben werden soll. Es handelt sich dabei nicht um eine Pfadangabe im klassischen Sinn, sondern um einen Bezeichner, der intern in den tatsächlichen Pfad der Ressource aufgelöst wird.

JSF sucht Ressourcen an den folgenden Stellen einer Anwendung (in der angegebenen Reihenfolge):

1. Im Wurzelverzeichnis der Webapplikation unter `/resources`
2. In `/META-INF/resources` im Classpath und somit auch in allen JARs

In unserem Fall versucht JSF die Bezeichnung `image.png` an einem der oben genannten Orte aufzulösen. Daher legen wir folgende Datei an:

```
/resources/image.png
```

Das oben angeführte Beispiel zeigt den einfachen Fall einer Ressource, die über ihren Namen referenziert wird. JSF 2.0 bietet aber darüber hinaus eine Einteilung in Bibliotheken, eine Versionierung und die Lokalisierung von Ressourcen.

Bibliotheken sind eine Möglichkeit, Ressourcen unter einem gemeinsamen Namen zu gruppieren. Der Bibliotheksname wird dann gemeinsam mit dem Ressourcennamen zur Identifikation der Ressource verwendet. Wir erweitern unser Beispiel um eine Bibliothek `images`, in der alle Bilder abgelegt sind. Die Codezeile von oben ändert sich wie folgt ab:

```
<h:graphicImage library="images" name="image.png"/>
```

JSF interpretiert die Bibliothek beim Auflösen der Ressource als zusätzliches Verzeichnis. Der Pfad der Bilddatei sieht folgendermaßen aus:

```
/resources/images/image.png
```

Doch damit noch nicht genug – Ressourcen und Bibliotheken können in mehreren Versionen existieren und lokalisiert werden. Die Funktionsweise zeigt Abschnitt 4.5.4.

Das Auflösen der Ressourcen und das Ausliefern der Daten an den Client übernimmt intern die Klasse `ResourceHandler`. Wie viele andere Teile von JSF kann auch der `ResourceHandler` über die `faces-config.xml` mit einer eigenen Implementierung dekoriert werden. Denkbar wären Implementierungen, die Ressourcen aus einer Datenbank holen oder dynamisch erzeugen – der Fantasie sind keine Grenzen gesetzt.

4.5.2 Ressourcen im Einsatz

Es gibt mehrere Wege, in JSF 2.0 Ressourcen in die Seite einzubinden. Folgende Standardkomponenten verfügen über die Attribute `name` und `library`, um direkt die auszugebende Ressource zu referenzieren:

- ❑ `h:graphicImage` gibt den Link auf ein Bild aus.
- ❑ `h:outputScript` gibt den Link auf ein Skript aus (ab JSF 2.0).
- ❑ `h:outputStylesheet` gibt den Link auf ein *Stylesheet* aus (ab JSF 2.0).

Ressourcen können auch direkt über einen EL-Ausdruck im Attribut `value` der entsprechenden Komponente referenziert werden – das implizite Objekt `resource` dient diesem Zweck. Eigenschaften dieses Objekts werden beim Auswerten des Ausdrucks als Ressourcenbezeichner interpretiert. Dieser Bezeichner kann der Name der Ressource oder der Name der Bibliothek gefolgt vom Namen der Ressource mit einem Doppelpunkt als Trennzeichen sein.

Hier nochmals das Beispiel aus dem letzten Abschnitt, diesmal allerdings über einen EL-Ausdruck realisiert:

```
<h:graphicImage value="#{resource['images:image.png']}" />
```

Kommt als VDL Facelets zum Einsatz, kann dieser EL-Ausdruck sogar direkt im HTML-Code verwendet werden:

```

```

Die dritte Methode zum Einbinden von Ressourcen erfolgt im Java-Code und ist vor allem für Entwickler von Komponenten interessant. Mit den beiden Annotationen `@ResourceDependency` und `@ResourceDependencies` können Abhängigkeiten zu Ressourcen bereits in der Komponenten- beziehungsweise Rendererklassendefinition definiert werden. Folgender Code verknüpft zum Beispiel eine Komponente mit der Ressource `script.js` aus der Bibliothek `scripts`:

```
@ResourceDependency(name="script.js", library="scripts")
public class MyComponent extends UIComponentBase {
    ...
}
```

Diese Methode wird hier nur der Vollständigkeit halber erwähnt. Weiterführende Informationen zum Thema Komponentenentwicklung und Ressourcen finden Sie in Abschnitt 5.2.3.

4.5.3 Positionierung von Ressourcen

Einen wichtigen Aspekt des Ressourcenmanagements von JSF haben wir bis jetzt noch nicht erwähnt. Stellen Sie sich vor, Sie benutzen eine Komponente aus einer Komponentenbibliothek, die ein spezielles Skript oder *Stylesheet* verwendet. Wie kommt diese Ressource dann in die Ansicht? Als Anwender der Bibliothek wollen wir uns nicht darum kümmern. Bei einigen Ressourcen wie *Stylesheets* und manchen Skripten kommt noch hinzu, dass sie an speziellen Stellen der Ansicht ausgegeben werden müssen, damit beim Benutzer die Seite richtig funktioniert. Wie die Verbindung zwischen Komponente und Ressource modelliert wird, haben wir bereits im letzten Abschnitt gezeigt. Das erklärt aber noch nicht, wie ein Link auf die Ressource in die Ansicht übernommen wird.

JSF erlaubt die Positionierung einzelner Ressourcen in definierten Bereichen der Ansicht wie dem Head oder dem Body. Damit JSF diese Bereiche richtig identifizieren kann, gibt es folgende neue Komponenten in der *HTML-Tag-Library*:

- ❑ `h:head` umschließt den Head-Bereich der Seite.
- ❑ `h:body` umschließt den Body-Bereich der Seite.

Beim Einbinden einer Ressource kann einer dieser Bereiche direkt adressiert werden. Dafür existiert in `@ResourceDependency` das Element `target` und in `h:outputScript` ein gleichnamiges Attribut. Die erlaubten Werte sind `head`, `body` und `form`, wobei *Stylesheets* allerdings unabhängig von der Angabe immer im Head ausgegeben werden (gemäß HTML-Standard müssen *Stylesheets* im Head-Bereich verlinkt werden, damit der Quelltext der Seite gültiges HTML bleibt).

Damit das Positionieren von Ressourcen funktioniert, ist es unbedingt nötig, `h:head` und `h:body` in allen Ansichten einzusetzen. Mit Facelets können Sie das in einem Template zentral für alle Seiten erledigen. Wie das funktioniert, erfahren Sie in Abschnitt 4.3.

Sehen wir uns anhand eines Beispiels an, wie das Positionieren von Ressourcen in der Praxis aussieht. Das folgende Dokument enthält im

Body-Bereich ein Stylesheet ohne Positionsangabe und ein Skript mit der Position head:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>
  <title>Ressourcen-Test</title>
</h:head>
<h:body>
  <h:outputStylesheet name="style.css"/>
  <h:outputScript name="test.js" target="head"/>
  <h:outputText value="Test"/>
</h:body>
</html>
```

Hier der gerenderte HTML-Code:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Ressourcen-Test</title>
  <link type="text/css" rel="stylesheet"
        href="/app/javax.faces.resource/style.css.jsf"/>
  <script type="text/javascript"
        src="/app/javax.faces.resource/test.js.jsf">
  </script>
</head>
<body>
  Test
</body>
</html>
```

Der Code zeigt, dass beide Ressourcen, das Skript und das Stylesheet, im Head-Bereich der Seite gerendert wurden. Mit Komponenten aus einer Bibliothek funktioniert das genauso – vorausgesetzt wird, dass der Entwickler die Komponente mit `@ResourceDependency` annotiert und das `target`-Attribut auf den Wert `head` gesetzt hat. Wenn Sie dann `h:head` und `h:body` auf der Seite einsetzen, erledigt JSF den Rest.

4.5.4 Identifikation von Ressourcen – Teil 2

In Abschnitt 4.5.1 haben wir zum Abschluss noch kurz darauf hingewiesen, dass JSF bei Ressourcen und Bibliotheken Versionierung und Lokalisierung unterstützt.

Versionsnummern sind bei Bibliotheken und Ressourcen mit durch Unterstrich (`_`) getrennte Zahlen wie `1_0` oder `1_0_1` angegeben. Sie werden, getrennt durch einen Schrägstrich (`/`), an den Namen der Resource oder der Bibliothek angehängt. Bei Ressourcennamen kann die

Version zusätzlich eine Dateierweiterung wie `.png` oder `.css` aufweisen. Warum das so ist, sehen wir gleich. Hier unser Beispiel mit Versionsnummern für die Bibliothek und die Ressource:

```
<h:graphicImage library="images/1_0"
  name="image.png/1_1.png"/>
```

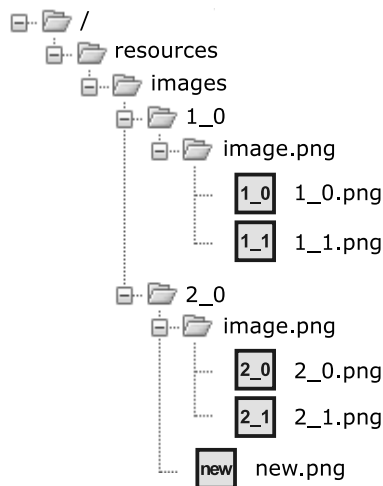
JSF interpretiert Bibliotheksversionen beim Auflösen als ein weiteres Verzeichnis im Pfad der Ressource. Ressourcenversionen werden dahingegen etwas anders behandelt. Existiert eine Ressource in mehreren Versionen, wird der Name der Ressource zu einem Verzeichnis, und die einzelnen Versionen sind die eigentlichen Ressourcendateien. Der neue Pfad der Bilddatei lautet dann folgendermaßen:

```
/resources/images/1_0/image.png/1_1.png
```

In den meisten Fällen ist es aber nicht nötig, die Versionsnummern beim Einsatz von Ressourcen zu definieren. Wenn keine expliziten Versionen angegeben sind, verwendet JSF automatisch die Ressource mit der höchsten Versionsnummer.

Um die verschiedenen Kombinationen von Ressourcennamen, Bibliotheksnamen und Versionsnummern zu veranschaulichen, wollen wir unser Beispiel erweitern. Abbildung 4.4 zeigt das Ressourcenverzeichnis der Anwendung mit einer Bibliothek und den zwei Ressourcen `image.png` und `new.png`. Links neben dem Namen der Bilddatei ist jeweils das Bild selbst dargestellt.

Abbildung 4.4
Beispiele von
Ressourcen



Basierend auf dem Verzeichnisbaum aus Abbildung 4.4 wollen wir nun verschiedene Ressourcen auflösen und uns das Ergebnis ansehen. Tabelle 4.1 zeigt für eine Reihe von Kombinationen aus Bibliotheks-

und Ressourcenname, welches Bild dadurch angezeigt wird. Beachten Sie bitte vor allem Namen ohne Versionsangaben und wie diese immer zur höchsten Version aufgelöst werden.

Bibliothek	Ressource	
images	image.png	2_1
images	image.png/2_1.png	2_1
images/2_0	image.png	2_1
images/2_0	image.png/2_1.png	2_1
images	image.png/2_0.png	2_0
images/1_0	image.png/1_0.png	1_0
images/1_0	image.png	1_1
images	new.png	new

Tabelle 4.1

Beispiele zur Auflösung von Ressourcen

Zum Schluss wollen wir noch die Lokalisierung von Ressourcen besprechen. JSF sucht beim Auflösen von Ressourcen nach folgendem Eintrag im *Application-Message-Bundle*:

```
javax.faces.resource.localePrefix=<Wert>
```

Falls dieser Eintrag für das aktuelle Locale gesetzt ist, wird dessen Wert als Teil des Pfads der Ressourcendatei interpretiert. Ist der Wert im deutschen Resource-Bundle beispielsweise auf `de` gesetzt, sieht der Pfad zu unserem Bild wie folgt aus:

```
/resources/de/images/image.png
```

Weiterführende Informationen zur Internationalisierung und zum *Application-Message-Bundle* finden sich in Abschnitt 2.14.

4.5.5 Ressourcen in *MyGourmet 12*

Die Umstellung von *MyGourmet* auf Ressourcen ist im momentanen Stand der Entwicklung trivial – wir erstellen dafür kein neues Beispiel.

Die wichtigste Änderung ist die Umstellung des Haupttemplates `template.xhtml` auf die Elemente `h:head` und `h:body`. Dafür müssen aber nur die entsprechenden HTML-Elemente ersetzt werden. Sobald das geschehen ist, können auch das Stylesheet und das Logo ins Verzeichnis `/resources` verschoben und als Ressourcen verwendet werden.

Die Ressourcenverwaltung wird erst ab dem nächsten Beispiel *My-Gourmet 13* interessant, wenn sich alles um die neuen Kompositkomponenten von JSF 2.0 dreht. Mehr dazu erfahren Sie in Abschnitt 5.1.

4.6 Die JSF-Umgebung: *Faces-Context* und *External-Context*

Immer wieder sind wir bisher an den unterschiedlichsten Stellen auf den *Faces-Context* gestoßen. Dieser Kontext wird in JSF durch die Klasse `javax.faces.context.FacesContext` repräsentiert. Der *Faces-Context* stellt die zentrale Schaltstelle einer JSF-Anwendung dar. Er wird ganz am Anfang jeder HTTP-Anfrage vom *Faces-Servlet* initialisiert und steht dem Entwickler ab dann als Parameter vieler Methoden, aber auch jederzeit über den Aufruf der Methode `FacesContext.getCurrentInstance()` zur Verfügung.

Die häufigste Anwendung des *Faces-Contexts* ist die Auflösung von *Managed-Beans* aus dem Quelltext einer Anwendung heraus. Listing 4.29 zeigt, wie der Zugriff auf eine *Managed-Bean* namens `personList` in einer anderen *Managed-Bean* aussieht. Es bietet sich an, solch syntaktische Grausamkeiten in eine Utility- oder eine Basisklasse auszulagern.

Listing 4.29

Zugriff auf eine
Managed-Bean im
Java-Code

```
FacesContext fc = FacesContext.getCurrentInstance();
fc.getApplication().getELResolver().getValue(
    fc.getELContext(), null, "personList");
```

Eine weitere wichtige Anwendung des *Faces-Contexts* ist das Hinzufügen von Nachrichten für die Darstellung auf der Webseite – und die Möglichkeit, auf die bisher hinzugefügten Nachrichten zuzugreifen. Zu diesem Zweck existieren folgende Methoden:

- ❑ `addMessage(String clientId, FacesMessage message):`
Fügt eine Nachricht zum *Faces-Context* hinzu. Falls eine Client-ID angegeben wird, bezieht sich die Nachricht auf die entsprechende Komponente, andernfalls ist sie global.
- ❑ `Iterator<FacesMessage> getMessages():`
Gibt einen Iterator über alle Nachrichten im *Faces-Context* zurück. Es sind auch jene inkludiert, die einer Komponente zugeordnet sind.
- ❑ `Iterator<FacesMessage> getMessages(String clientId):`
Gibt einen Iterator über alle Nachrichten im *Faces-Context* für die Komponente mit der angegebenen Client-ID zurück.

- ❑ `List<FacesMessage> getMessageList():`
Gibt eine Liste mit allen Nachrichten im *Faces-Context* zurück (ab JSF 2.0). Es sind auch jene inkludiert, die einer Komponente zugeordnet sind.
- ❑ `List<FacesMessage> getMessageList(String clientId):`
Gibt eine Liste über alle Nachrichten im *Faces-Context* für die Komponente mit der angegebenen Client-ID zurück (ab JSF 2.0).

Weiterführende Details zum Hinzufügen und Verwalten von Nachrichten finden Sie in Abschnitt 2.13.

Über den *Faces-Context* gelangen Sie auch zur *Application*, und die hilft Ihnen sowohl beim Erzeugen von neuen Komponenten als auch von *Method-* und *Value-Expressions* (siehe Listing 4.30).

Applikationsobjekte anlegen

```
fc.getApplication().getExpressionFactory().
    createValueExpression(ELContext ctx,
        String expression, Class expectedType);
```

Listing 4.30
Applikationsobjekte anlegen

```
fc.getApplication().getExpressionFactory().
    createMethodExpression(ELContext ctx, String expression,
        Class expectedReturnType, Class[] params);
```

```
fc.getApplication().createComponent(String componentType);
```

Wobei der *ELContext* – wie in Listing 4.29 bereits gezeigt – als Eigenschaft des *Faces-Contexts* verfügbar ist. Auch hier zahlt es sich aus, Utility-Methoden für das Erzeugen neuer Elemente vorzusehen.

Ein wichtiger Bereich im *Faces-Context* ist die Möglichkeit, den Lebenslauf einer HTTP-Anfrage zu beeinflussen. Dazu gibt es folgende Methoden, die bereits bei der Ereignisbehandlung in Abschnitt 2.8 zum Einsatz gekommen sind:

Lebenslauf beeinflussen

- ❑ `renderResponse():`
Nach Abschluss der momentan laufenden Phase wird sofort die HTTP-Antwort gerendert (die Ausführung des Lebenszyklus springt sofort zur Render-Response-Phase).
- ❑ `responseComplete():`
Die Abarbeitung des Lebenszyklus wird nach Abschluss der momentan laufenden Phase beendet.

Schließlich bietet der *Faces-Context* noch die Möglichkeit, auf den *External-Context* zuzugreifen. Der *External-Context* ist der Wrapper rund um die der Webanwendungs-Umgebung zugrundeliegende Funktionalität; also in den meisten Fällen um entweder den *ServletContext* oder *PortletContext*. Hier der für einen Zugriff notwendige Code:

External-Context

```
FacesContext fc = FacesContext.getCurrentInstance();
ExternalContext ec = fc.getExternalContext();
```

Auch der *External-Context* bietet einige interessante Methoden, die sich aus der Funktionalität des *Basis-Contexts* ergeben. Hier eine selektive Auswahl:

- ❑ `Map<String, Object> getRequestMap()` liefert eine veränderbare *Map* mit allen Attributen des Requests zurück. Dazu zählen unter anderem auch alle instanziierten *Managed-Beans* im Gültigkeitsbereich `request`, nicht aber die Request-Parameter.
- ❑ `Map<String, String> getRequestParameterMap()` liefert eine nicht veränderbare *Map* mit allen Request-Parametern zurück. Der Name des Parameters ist dabei der Schlüssel der *Map*.
- ❑ `Map<String, Object> getSessionMap()` liefert eine veränderbare *Map* mit allen Attributen der Session zurück. Dazu zählen unter anderem auch alle instanziierten *Managed-Beans* im Gültigkeitsbereich `session`.
- ❑ `String getRemoteUser()` liefert den Namen des Benutzers zurück, der den Request abgesetzt hat, falls dieser vorher authentifiziert wurde.
- ❑ `boolean isUserInRole(roleName)` prüft Rollenberechtigungen und liefert `true` zurück, wenn der authentifizierte Benutzer für die Rolle `roleName` autorisiert ist.

Der *External-Context* bietet noch einige weitere Methoden, die Sie am besten der API-Dokumentation entnehmen.

4.7 Konfiguration von JavaServer Faces

In den bisherigen Kapiteln wurden die Konfigurationsmöglichkeiten von JSF bereits in Beispielen behandelt, im kommenden Abschnitt sehen wir uns die Konfiguration ein wenig detaillierter an.

JSF verwendet ein Minimum an zu editierenden XML-Dateien. Sieht man von einer eventuellen Verwendung von *Tiles* oder *Portlets* ab, existieren nur zwei Konfigurationsdateien für eine JSF-basierte Webanwendung:

- ❑ *Webanwendung-Konfigurationsdatei web.xml*:
Wie bei allen J2EE-Webapplikationen dient dieser *Deployment-Deskriptor* zum Setzen zentraler Einstellungen in der Applikation. Hier finden sich Definitionen über Kontextparameter, Filter, das *Faces-Servlet* oder auch das *Servlet-Mapping* wieder.

- ❑ *JSF-Konfigurationsdatei faces-config.xml*:
Diese XML-Datei ist die zentrale Konfigurationsdatei von JSF. In ihr deklariert man Managed-Beans, bestimmt Navigationsregeln, behandelt die Internationalisierung der Applikation und legt weitere JSF-Spezifika fest.

Diese beiden müssen im Verzeichnis WEB-INF der Applikation vorhanden und fehlerfrei sein, sonst lässt sich die Webanwendung im Applikationsserver oder *Servlet-Container* nicht hochfahren.

4.7.1 Die Webkonfigurationsdatei web.xml

Sehen wir uns nun in weiterer Folge eine typische web.xml-Datei einer JSF-Applikation genauer an. Der wichtigste Teil des Deployment-Deskriptors ist die Spezifikation des *Faces-Servlets*, das die Anfragen an die JSF-Anwendung bearbeitet, und dessen *Mapping*. Ein weiterer wichtiger Aspekt der Webapplikation, der in diesem Abschnitt behandelt wird, sind Konfigurationsparameter.

Faces-Servlet und Mapping

Jede JSF-Applikation muss ein Faces-Servlet konfigurieren. Listing 4.31 zeigt nochmals die web.xml-Datei des Beispiels *MyGourmet 1*, in der selbstverständlich auch ein Servlet und ein zugehöriges Servlet-Mapping definiert sind.

Als Faces-Servlet können (in aufsteigender Schwierigkeitsstufe der Verwendung) folgende Klassen eingesetzt werden:

- ❑ Eine direkte Referenz auf die von JSF mitgelieferte Klasse `javax.faces.webapp.FacesServlet`
- ❑ Eine von `org.apache.myfaces.webapp.MyFacesServlet` abgeleitete Klasse
- ❑ Eine völlig frei implementierte Klasse

Implementieren Sie das Faces-Servlet frei, sollten Sie zumindest den Faces-Context initialisieren und den Lebenslauf starten. Befinden sich in der web.xml mehrere Servlets, gewinnt die Einstellung im Element `load-on-startup` an Bedeutung. Hier ist es möglich, die Reihenfolge der beim Start initialisierten Servlets zu bestimmen.

Welche Anfrage auf welches Servlet weitergeleitet wird, zeigt das `servlet-mapping`-Element an. Der Wert des `url-pattern`-Elements definiert ein Präfix oder Postfix der Anfrageadresse, das dem *Faces-Servlet* zugewiesen wird. Im Beispiel *MyGourmet 1* wurde die Postfix-Zuordnung (auch *Extension-Mapping* genannt) `*.jsf` gewählt, wobei

Listing 4.31

Deployment-Deskriptor
von MyGourmet 1

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns="http://java.sun.com/xml/ns/j2ee"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
5         http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd"
6     version="2.5">
7     <description>JSF@Work - MyGourmet 1</description>
8     <servlet>
9         <servlet-name>Faces Servlet</servlet-name>
10        <servlet-class>
11            javax.faces.webapp.FacesServlet
12        </servlet-class>
13        <load-on-startup>1</load-on-startup>
14    </servlet>
15    <servlet-mapping>
16        <servlet-name>Faces Servlet</servlet-name>
17        <url-pattern>*.jsf</url-pattern>
18    </servlet-mapping>
19    <welcome-file-list>
20        <welcome-file>index.jsp</welcome-file>
21        <welcome-file>index.html</welcome-file>
22    </welcome-file-list>
23 </web-app>
```

sämtliche Anfragen mit der Endung `.jsf` durch das *Faces-Servlet* gehandhabt werden. Intern wird diese Adresse dann auf die View-ID der Seitendefinition umgelegt.

Die zweite Möglichkeit, ein *Servlet-Mapping* zu definieren, ist eine Präfix-Zuordnung `<url-pattern>/faces/*</url-pattern>`. Die View-ID der Seitendefinition ergibt sich in diesem Fall direkt aus der URL, nachdem das Präfix entfernt wurde.

Sehen wir uns kurz den entscheidenden Unterschied zwischen Präfix- und Postfix-Mapping anhand eines Beispiels an. Wir wollen dazu die Seite mit der Seitendefinition `/helloWorld.jsp` der fiktiven Webanwendung `http://www.mustermann.org` im Browser laden. Mit einem Postfix-Mapping müssen wir dazu die URL

```
http://www.mustermann.org/helloWorld.jsf
```

in die Adressleiste eingeben. Mit einem Präfix-Mapping sieht die URL folgendermaßen aus:

```
http://www.mustermann.org/faces/helloWorld.jsp
```

In der Regel sind beide Methoden für eine JSF-Anwendung einsetzbar. In manchen Fällen, wie beim Einsatz von Tomahawk oder Trinidad, kann es allerdings von Vorteil sein, ein Präfix-Mapping zu verwenden.

Kontextparameter

An erster Stelle in der Konfiguration stehen üblicherweise die Kontextparameter. Die optionale Angabe von `context-param`-Elementen dient der Definition von Parametern zur Initialisierung des *Servlet-Contexts*. Hier können Basiseinstellungen vorgenommen werden. Die folgende Liste zeigt eine Übersicht der wichtigsten Einstellungen für JSF:

- ❑ `javax.faces.CONFIG_FILES`:
Über `param-value` können JSF-Konfigurationsdateien angegeben werden. Auch wenn der Wert dieses Parameters leer bleibt, wird die Standarddatei `WEB-INF/faces-config.xml` immer geladen, daher ist der Parameter optional.
Für eine bessere Übersichtlichkeit kann es auch wünschenswert sein, mehrere solcher Dateien einzuführen. Die Namen dieser Dateien müssen dann im `param-value`-Element durch Kommata voneinander getrennt werden. Listing 4.32 zeigt ein Beispiel mit einer jeweils eigenen Konfigurationsdatei für Managed-Beans, Navigationsregeln und die Internationalisierung der Applikation.

```
<param-name>javax.faces.CONFIG_FILES</param-name>
<param-value>
    /WEB-INF/faces-beans.xml,
    /WEB-INF/faces-navigation.xml,
    /WEB-INF/faces-I18N.xml
</param-value>
```

Listing 4.32

Einsatz mehrerer Konfigurationsdateien

- ❑ `javax.faces.DEFAULT_SUFFIX`:
Dieser Parameter definiert eine durch Leerzeichen getrennte Liste von Erweiterungen für View-Identifizier, die JSF als JSP-Deklarationen interpretieren soll. Der Standardwert ist `.jsp`.
- ❑ `javax.faces.FACELETS_SUFFIX`:
Dieser Parameter definiert eine durch Leerzeichen getrennte Liste von Erweiterungen für View-Identifizier, die JSF als Facelets-Deklarationen interpretieren soll. Der Standardwert ist `.xhtml`.
- ❑ `javax.faces.FACELETS_VIEW_MAPPINGS`:
Dieser Parameter definiert eine durch Semikolons getrennte Liste von View-Identifiern, die JSF als Facelets-Deklarationen interpretieren soll. Diese Liste kann auch Einträge mit Wildcards wie `/secure/*` enthalten.
- ❑ `javax.faces.PROJECT_STAGE`:
Mit diesem Parameter kann die Projektphase der Anwendung auf einen der folgenden Werte gesetzt werden: `Production` (Standardwert), `Development`, `SystemTest` oder `UnitTest`.

- `javax.faces.STATE_SAVING_METHOD`:
JSF speichert beim Rendern einer Ansicht den Zustand des Komponentenbaums für nachfolgende Anfragen auf dieselbe Ansicht. Die Datenhaltung kann entweder auf dem Client oder auf dem Server geschehen. Ist `param-value` auf `server` gesetzt, verringert sich der Bandbreitenbedarf, dafür wird der Server stärker unter Last gesetzt. Steht `param-value` hingegen auf `client`, steigt das Datenaufkommen und der Komponentenbaum muss clientseitig in `<input type="hidden"/>`-Felder serialisiert (*Base64-encoded*) werden. Ist kein ausdrücklicher Grund vorhanden, den Status der Applikation auf dem Client zu speichern, wird empfohlen, die Standardmethode `server` nicht zu überschreiben.

4.7.2 Die JSF-Konfigurationsdatei – `faces-config.xml`

In diesem Abschnitt geht es darum, zentrale Einstellungen für *JavaServer Faces* zu treffen. Einerseits bietet die `faces-config.xml` die Möglichkeit, alltägliche Konfigurationseinstellungen in der Entwicklung einer JSF-Applikation zu setzen. Dazu gehört die Konfiguration von *Managed-Beans*, Navigationsregeln sowie Einstellungen zur Applikation selbst. Der zweite Aufgabenbereich, die Registrierung von Komponenten, Renderern, Validatoren und Konvertern, ist für den Entwickler von Komponenten (und Komponentenbibliotheken) interessant. Zuletzt gibt es noch erweiterte Möglichkeiten, wie das Einbinden von *Phase-Listenern* und die Konfiguration von *Factories* für die Erzeugung von JSF-Kernklassen.

JSF 2.0 bietet für die meisten Einstellungen in der Konfigurationsdatei `faces-config.xml` alternativ die Möglichkeit, Annotationen einzusetzen.

Die Konfiguration von *Managed-Beans* erfolgt in jeweils einem `managed-bean`-Element pro Bean oder ab JSF 2.0 mit der Annotation `@ManagedBean`. Eine genauere Beschreibung der umfassenden Einstellungsmöglichkeiten findet sich in Abschnitt 2.4.2.

Die Regeln für die Navigation in einer JSF-Anwendung werden mithilfe von `navigation-rule`-Elementen definiert. Wie das genau funktioniert, zeigt Abschnitt 2.7. Im Laufe der Entwicklung einer JSF-Webapplikation können sehr viele Navigationsregeln anfallen. Um die Übersicht zu behalten, empfiehlt sich die Auslagerung einzelner Teile der Konfiguration in separate Dateien, wie es im Abschnitt der JSF-Konfigurationsdatei `faces-config.xml` beschrieben wurde.

Anwendungseinstellungen – application

Ein zentrales Element in der `faces-config.xml` ist das Tag `application`. Darin werden wichtige Einstellungen für Kernbereiche von JSF getroffen. Das Element `application` kann als die zentrale Schaltstelle einer JSF-Anwendung betrachtet werden. Hier ist es möglich, neben Einstellungen zur Lokalisierung und der Definition von Message- und Resource-Bundles, essenzielle Teile wie den View-Handler oder den EL-Resolver von JSF mit eigenen Implementierungen zu dekorieren. Damit wird dem Benutzer ein mächtiges Werkzeug in die Hand gegeben, um die Applikation an eigene Bedürfnisse anzupassen.

Listing 4.33 zeigt ein Beispiel für eine `faces-config.xml`, auf deren Elemente wir weiter unten eingehen werden.

```
1 <faces-config>
2 <application>
3   <navigation-handler>
4     at.irian.jsfatwork.MyNavigationHandler
5   </navigation-handler>
6   <el-resolver>
7     org.springframework.web.jsf.el.SpringBeanFacesELResolver
8   </el-resolver>
9   <message-bundle>
10    at.irian.jsfatwork.messages
11  </message-bundle>
12  <locale-config>
13    <default-locale>de</default-locale>
14    <supported-locale>en</supported-locale>
15  </locale-config>
16  <resource-bundle>
17    <base-name>at.irian.jsfatwork.text</base-name>
18    <var>text</var>
19  </resource-bundle>
20 </application>
21 </faces-config>
```

Listing 4.33

`faces-config.xml`
mit *Spring-EL-Resolver*

In der folgenden Auflistung finden Sie eine Übersicht der wichtigsten Einstellungen. Bei den einzelnen Punkten handelt es sich jeweils um Kindelemente des Tags `application`:

- ❑ `locale-config` ist ein Container zur Definition der von der Anwendung unterstützten Sprachen. In Listing 4.33 ist zum Beispiel Deutsch als Standardsprache und Englisch als unterstützte Sprache definiert. Die Konfiguration der Internationalisierung wird ausführlicher in Abschnitt 2.14 behandelt.

- ❑ `message-bundle` definiert den Namen des Resource-Bundles, das die Nachrichten der Applikation enthält, wie Listing 4.33 zeigt. Genauere Informationen hierzu finden Sie in Abschnitt 2.14.2.
- ❑ `resource-bundle` ist ein Container für die Definition eines Resource-Bundles, das Texte für die Anwendung enthält. Im Beispiel in Listing 4.33 wird das Resource-Bundle `at.irian.jsfatwork.text` unter dem Namen `text` in Seitendefinitionen verfügbar gemacht. Genauere Informationen hierzu finden sich in Abschnitt 2.14.3.
- ❑ `navigation-handler` definiert die Klasse des verwendeten *Navigation-Handlers*. Dieser kann zum Beispiel für erweitertes Logging oder andere spezielle Zwecke im Navigationsbereich überschrieben werden.
- ❑ `view-handler` definiert die Klasse des verwendeten *View-Handlers*, für den es eine Grundimplementierung gibt. Im Fall der Verwendung von Facelets in JSF vor Version 2.0 müssen Sie hier die Klasse des Facelets-View-Handlers eintragen (`com.sun.facelets.FaceletViewHandler`).
- ❑ `state-manager` definiert die Klasse des verwendeten *State-Managers*.
- ❑ `el-resolver` definiert die Klasse des verwendeten *EL-Resolvers*, der für die Auflösung der *Value-Expressions* zuständig ist. Hier muss unter anderem dann eine neue Klasse eingetragen werden, wenn das *Spring-Framework* zur Verwaltung der *Managed-Beans* eingesetzt werden soll. Listing 4.33 zeigt eine Konfigurationsdatei für eine Anwendung, die *Spring* in einer Version ab 2.5 einsetzt. Die Art der Einbindung in der `faces-config.xml` eines JSF-Projekts ist sowohl von der Version von JSF als auch von der Version von *Spring* abhängig. Mehr dazu in Abschnitt 7.1.

Die weiteren möglichen Elemente der `faces-config.xml`, wie die Registrierung der Renderer, Konverter oder Validatoren, werden in den Abschnitten 2.11 und 2.12 beziehungsweise im Kapitel 5 (für die Konfiguration von Komponenten) noch genauer behandelt.